CONLAN-a formal construction method for hardware description languages: basic principles

R. Piloty, M. Barbacci, D. Borrione, D. Dietmeyer, F. Hill, P. Skelly

CONLAN—A formal construction method for hardware description languages: basic principles

by ROBERT PILOTY
Technische Hochschule Darmstadt, FR Germany

MARIO BARBACCI
Carnegie-Mellon University

DOMINIQUE BORRIONE
Universite de Grenoble, France

DONALD DIETMEYER
University of Wisconsin-Madison

FREDRICK HILL
University of Arizona-Tucson

and

PATRICK SKELLY
Honeywell, Phoenix

1. INTRODUCTION

The development of a CONLAN(CONsensus LANguage) goes back to the first Symposium on Hardware Description Languages (HDL) at Rutgers University in 1973. It was initiated by J. Lipovski, then Univ. of Florida. After two years of preparatory work the CONLAN Working Group was formed on the occasion of the third Symposium on HDL in New York. These papers represent the result of four years of hard work of a group spread out over two continents. This work is by no means complete; many things have still to be done. Nevertheless, encouraged by the positive response to an informal presentation of our approach at the fourth Symposium on HDL in Palo Alto 1979, we feel that publication of what we have obtained so far is warranted. This paper presents the basic principles of CONLAN. Two companion papers [1,2] treat language derivation and language application within the framework of CONLAN. A more detailed report, of which a draft exists already, will be forthcoming soon.

2. MOTIVATION AND OBJECTIVES

The decision to start the CONLAN project was motivated by the following assessment of the situation in the area of HDL and of Computer Aided Design (CAD) tools based on them:

Several dozen HDL's existed in 1973 [3] and every year since new languages have been proposed and published, mostly from persons in academic institutions [4,5,6]. This tendency to proliferation is in sharp contrast to acceptance in industry. Neither have they been used to document the design process of digital systems nor to support tools for certification, synthesis, and performance evaluation to any appreciable extent. Most CAD tools in industry are designed to aid the manufacturing process (placement, routing, mask layout etc.). The process of systems and logic design is mostly carried out in the traditional way of drawing block and circuit diagrams at the IC package or gate level. In many cases these diagrams are the only true and complete documentation of the system. Most other aspects or phases of the system design, particularly system behavior, are informally and incompletely described. Simulation as a means for advanced certification is used, if at all, at a very low level (mostly gate level) and hence at enormous cost for more complex systems. Most of the certification is done very late at the level of a physical prototype causing costly changes in physical design.

This situation has not changed very much in the four years of CONLAN development: HDL's continue to proliferate [7,8] but their usage in real life design has not increased in the same proportion. Only recently a growing interest in efficient tools for design support at systems and logic level can be observed, probably due to the advance of LSI and VLSI, where late changes make a system more and more costly, and due to increased system complexity in a competitive market, calling for more efficient design tools [9,10].

There are several reasons why acceptance of existing HDL's is so low:

1. None of the languages alone is of sufficient scope to portray all aspects of a system and cover all phases of the design process.
2. Languages of different scope are syntactically and semantically unrelated.
3. Few of the languages are formally defined.
4. Only a few languages are implemented.
5. Descriptions are represented by character strings rather than diagrams.
6. There exists no comprehensive hardware and firmware design methodology telling how to use HDL's effectively.

The main aim of the CONLAN Working Group is to remedy the first four deficiencies [11]. Its primary objectives are:

1. to provide a common formal syntactic and semantic base for all levels and aspects of hardware and firmware description, in particular for descriptions of system structure and behavior.
2. to provide a means for the derivation of user languages from this common base
   —having a limited scope adjusted to a particular class of design tasks,
   —thus being easy to learn and simple to handle,
   —yet having a well defined semantic relation among each other.
3. to support CAD tools for documentation, certification, design space exploration, synthesis and so on.

CONLAN is not intended as a language standard trying to impose a certain style of hardware description on makers of design tools. It should rather be viewed as a formal system which allows them to construct HDL's of their choice in a consistent and unambiguous way within some notational conventions.

Character strings using the ISO-IRV 646 character set are the basic means of representing CONLAN descriptions, since they are more general and easier to use as an input or output for CAD tools. However it is an objective of the CONLAN Group at a later stage, to show how to write descriptions in CONLAN which are isomorphic to space structure (network descriptions) or behavior (sequential and concurrent flow descriptions), and to propose graphical members of the CONLAN family.

Problems of design methodology are not treated in this presentation of CONLAN. Much remains to be done in this area although a number of significant results have been obtained already.

3. BASIC APPROACH

Language family

CONLAN supports a *self-defining, extensible* family of languages. Its member languages are tied together by a common core syntax and a common semantic definition system. The CONLAN construct to define a member language is called a *language definition segment.*

Descriptions

The member languages are used to write descriptions of hardware, firmware or software modules. *Description definition segments* are provided for this purpose as a CONLAN construct.

Abstract datatypes

The method for semantic definition is based on the concept of abstract data types, which has been developed for programming languages like CLU [12,13] or ALPHARD [14,15]. An abstract data type, henceforth called a *type*, is defined by a domain of elements and a set of operations on these elements. New types may be defined in terms of primitive types supplied with the language.

Reference language

Contrasting the application of types in programming languages, the primitive types of a CONLAN member language except Primitive Set CONLAN (described later in this paper) are not implied. Rather, they are defined in terms of the types of one other member language. This language is called the *reference language* of the language being defined. This establishes a partial order among the member languages: A language L_b is derived from a language L_a if L_a is the reference language of L_b or if there is a chain of reference languages leading from L_b to L_a.

Self-definition

In CONLAN the same construction mechanism and the same notational system used to provide descriptions is also used to define new language members. In this sense CONLAN is self-defining in contrast to externally defined languages using a separate language to define its semantics, e.g., the formal description of PL/I using the Vienna Definition Language [16].

Extensibility

The CONLAN family of languages is open ended. New languages may be derived from existing ones at any given point in time as the need arises. They in turn may be used later as reference languages for further languages.

Syntax modification—core syntax

The syntax of a new CONLAN member may be made to differ from the syntax of its reference language by adding

and/or deleting productions in the reference syntax. A FORMAT statement is provided for this purpose. This capability permits the language designer to keep the syntax of a new language as simple as possible and yet allows the incorporation of new constructs to denote specific features, e.g., the introduction of an infix symbol to denote some new operation. There is a set of productions which may never be deleted, and thus is common to all CONLAN members. It is called the *core syntax*.

CONLAN text structure

The CONLAN text structure is shown in Figure 1. At any given point in time it consists of a set of language definition segments and a set of description segments. Each segment is under the scope of a REFLAN statement. In a language definition segment this statement points to the language from which the new language is directly derived. Thus language LL1 is directly derived from L1, and LL2 is directly derived from L2. In a description segment the REFLAN statement points to the language in which the description is written.

Hiding of types and operations

It is important to note the CONLAN concept of hiding types and operations.

Referring to Figure 1, for the writer of a LL1 only those types and operations which are defined in L1 and not marked PRIVATE are visible and accessible. This implies that the types and operations of bcl are inaccessible to LL1 unless explicitly brought forward by L1 with a CARRY statement. The CARRY statement avoids the need for redefinition if a type is used in more than one language level. This hiding mechanism is the main instrument to keep derived languages simple and maintain a clear semantic relation with their ancestor languages.

Base conlan—toolmakers, users

There is one root language called base conlan (bcl), serving as the interface between the CONLAN Working Group and its public of *toolmakers* and *users*. Toolmakers start from bcl to construct languages and their associated CAD tools. Users write descriptions of hardware and firmware systems in these languages. Bcl provides a carefully chosen set of basic types reflecting the CONLAN concept of time and space, of signals and carriers, of arrays and records. It is described in more detail in Reference 1.

Primitive set conlan

To define bcl the CONLAN Working Group used a very low level but powerful language called primitive set conlan (pscl) to formally define the concepts represented by the bcl types. Pscl has no reference language. It owns a set of primitive types whose domains and operations are introduced informally. The rest of this paper is devoted to the formal concepts of CONLAN and a description of pscl.

4. FORMAL CONCEPTS

CONLAN deals with the following categories of objects

—Elements, Parameters, Operations, Types and Classes, Descriptions, and Languages

Identifiers are used to denote CONLAN objects. *Simple identifiers* start with at least one letter followed by an arbitrary string of letters, digits, and underscore ('_'), of any length and terminating with a letter, or digit, or with the symbol @. Symbol @ denotes a system identifier which may be used only within a language definition segment. *Compound identifiers* are two or more simple identifiers separated by period ('.'). They permit one to prefix a simple identifier with one or more enclosing segment identifiers if the corresponding object is referenced outside the segment providing the simple identifier.

4.1 Elements

Elements are the operands for CONLAN operations. CONLAN works with a well defined universe of elements. Subsets of this universe serve as domains for the operations.

4.2 Parameters

There are formal and actual parameters. A formal parameter is denoted by an identifier representing any element of a given type. Formal parameters appearing in a parameter list of an operation or type or a description are typed, i.e., the parameter is followed by the type designator separated by a colon (e.g., x:bool). The designator defines the domain of x and the operations applicable to it.

Typing of formal parameters permits type checking: When a formal parameter is bound to an actual parameter, its associated type designator is used to check if the type of the actual parameter is equivalent to the type of the formal parameter. The type of the actual parameter is normally explicitly stated in a declaration, or in one of the constructs involving predicates (e.g., parameter a in ALL a:int WITH pred(a,u,v) ENDALL). On the other hand, when the actual parameter is a constant denotation, it can also be checked to determine that it belongs to the domain of the prescribed type.

Generic segments [17] may be specified in CONLAN, i.e., segments which accept an operation identifier (operator symbol) or a type designator as a parameter. Formal generic parameters must be typed using keyword FUNCTION or ACTIVITY or a class designator. Consider for example:

x(.., f:FUNCTION(int,int):bool,..) or x(..,u:someclass,..)

In the first case f is typed to accept any function identifier which is defined as a binary function with an integer domain

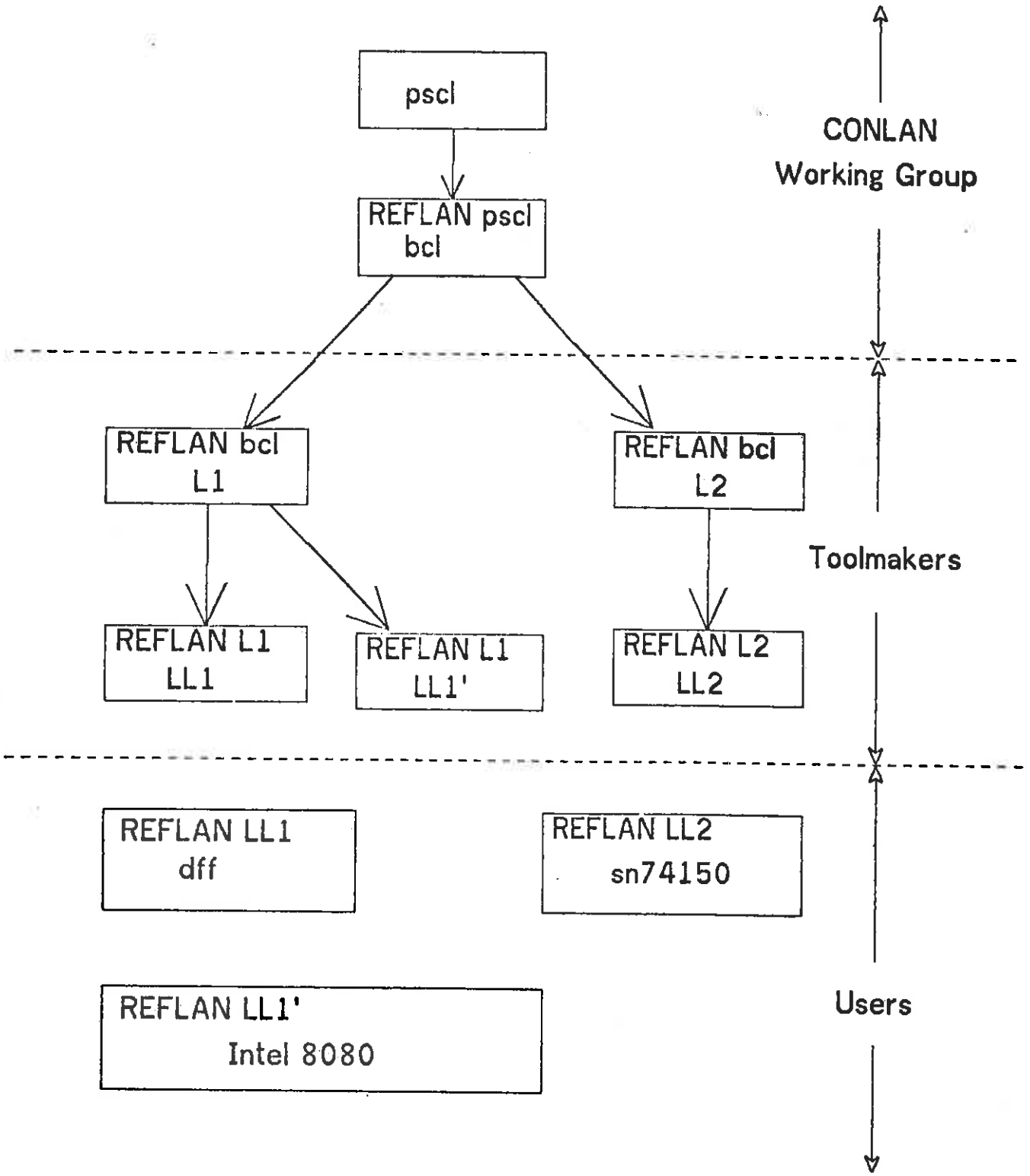

Figure 1—CONLAN text structure.

and a Boolean range. In the second case *u* accepts any type designator, which belongs to the class 'someclass.'

4.3 Operations

CONLAN operations are normally denoted by an identifier prefixing a list of parameters. Operator symbols may be introduced for prefix and infix notation either in lieu of a standard functional notation or as an alternative to it via syntax modification.

Two categories of operations are known in CONLAN: *activities* and *functions*.

Activities

An activity changes the state of one or more carriers. It is defined by an *activity definition segment* of the form:

ACTIVITY ident (typed-parameters) BODY.....ENDident

Formal parameters are typed. The body may contain declarations for local carriers and enclose a non-empty list of invocations of previously defined activities.

An activity may be invoked in five modes:

—unconditional, e.g.: act1(exp1,exp2,...)
—IF c1 THEN act1 ELIF c2 THEN act2....ELSE actn ENDIF
—ON c1 IS act1 ENDON
—CASE exp IS x1:act1,....x_n:actn, ELSE act ENDCASE
—ONCASE exp IS x1:act1,....x_n:actn, ELSE act ENDON

If invoked under IF or CASE it is evaluated as long as the condition is true. If invoked under ON or ONCASE it is evaluated whenever a change in the value of the condition is detected.

Functions

A function maps a domain of elements into a range of elements. It is defined by a *function definition segment* of the form:

FUNCTION ident(typed-parameters): result-type
BODY....RETURN expr ENDident

Formal parameters are typed. The body may contain declarations for local carriers and enclose a non-empty list of invocations of previously defined activities.

Functions are normally invoked in expressions. Expressions returning a Boolean result may also be called predicates. A new expression may be formed from existing ones in one of the following forms.

—nesting, e.g. f1(exp1,exp2,...)
—IF c1 THEN exp1 ELIF c2 THEN exp2.... ELSE expn ENDIF
—THE@ r:restype WITH@ pred(*r*,*a*,*b*...) ENDTHE

The last form selects the element *r* from the result type for which the predicate becomes true. If it becomes true for none or several elements of restype an error condition exists.

To construct new predicates from existing predicates the equivalent of quantifiers known from first order predicate calculus is available in CONLAN. The following forms may be used in addition to those of the preceding paragraph:

—FORALL@ x:atype IS@ pred(*x*,*a*,*b*....) ENDFOR
—FORSOME@ x:atype IS@ pred(*x*,*a*,*b*....) ENDFOR
—FORONE@ x:atype IS@ pred(*x*,*a*,*b*....) ENDFOR

Assertions are predicates which denote conditions on actual input, output and attribute parameters as well as on local objects. Assertions must be true at every point in time when the segment containing these assertions is invoked (operations) or used (descriptions). If an assertion is not true an error condition exists.

4.4 Types and classes

Types are defined in *type definition segments* of the following form (simplified):

TYPE t2(typed-parameters) BODY set-definition carry operation-definition ENDt2

Parameters are optional. If present, a *family of types* is specified by the definition.

The set definition part serves to specify the domain of the type. It may be specified in any of three ways:

1. as a true subset of the domain of an existing type t1 using the subset constructor:

   ALL x:t1 WITH pred(*x*,*a*,*b*) ENDALL

2. identical to the domain of an existing type by simply writing its designator, t1;
3. by enumeration of constant denotations.

In cases (1) and (2) type t1 is called the *defining type*. In case (3) the defining type is the universal type univ@.

In the operation definition part, operations may be defined to operate on the new type. These operations are defined in terms of the operations previously defined including operations of the defining type t1. The operations of the defining type t1 may not be applied to the domain of the defined type t2 outside the body of the definition of t2 (hiding of operations) unless explicitly listed in the carry part. The parameters or results of the carried operations originally typed with t1 are then considered typed with t2 (implicit type conversion).

Type designators, may be used as operands in type relations:

1. Two types s, t are *equal* ($s=t$) if s and t refer to the same type definition segment.
2. Let s and t designate two types. Then t is *derived from* $s(t<|s)$ iff $s=t$ or t is a subtype of s or s is the defining type of t or there exists a set of types $t_m, t_{m-1},\ldots,t_i,\ldots, t_1$, for $m>2$ such that $t_m=t$ and $t_1=s$ and $t_i<|t_{i-1}$ for all $i=2,\ldots,m$.
3. Let x represent an element of the CONLAN universe and t the designator of any type in CONLAN. Then x is an *element of* $t(x\epsilon t)$ if x is the denotation of an element of the domain of t.

Classes are named sets of type designators together with operations defined on elements of that set. A class is defined by a *class definition segment* analogous to a type definition. The class whose domain consists of the type designators of all types defined or yet to be defined is called the universal class any@. The main use of class definitions is to introduce designators for generic parameters, e.g., TYPE array (u:any@).

4.5 Descriptions

A *description definition segment* is used to define the input/output relation of hardware, firmware, or software modules. Details about their definition and usage can be found in Reference 2.

DESCRIPTION nand(IN x,y: btml OUT z: btml) BODY $z.=(x\text{-}\wedge y)\Delta 1$ ENDnand

In the above example, Δ is the decay operator, .= denotes terminal connection, and btm1 has been defined as the type "Boolean terminal with default value 1" in Reference 1.

4.6 Languages

A CONLAN member language is defined via a *language definition segment* of the form:

REFLAN oldlanguage CONLAN newlanguage
BODY...ENDnewlanguage

The constituents of the body are:

—list of carried types and operations from the reference language (optional)
—definition of public and private types
—operation definitions (optional)
—description definitions (optional)
—format statements (optional)

Notice that operations may be defined outside type segments. A library of operations could therefore be defined as part of a CONLAN segment. The same applies to descriptions if the language is geared to describing systems in terms of standard modules (e.g. TTL-modules). Syntax modifications via format statements will be illustrated in Reference 1.

All types referenced in the formal parameter lists appearing in non-private type, operation, and description definitions must either be explicitly defined or carried from the reference language (closure of a language with respect to types).

5. PRIMITIVE SET CONLAN (pscl)

Pscl is the lowest language level in CONLAN. It has no reference language. Hence its types (domains and operations) are defined informally.

UNIV@

Type **univ@** consists of all members of all types defined or yet to be defined in all members of the CONLAN family, together with operators '=' and '≠'. It permits the present definition of operations on objects to be defined in the future:

univ@={.., −1, 0, +1,.., '!',.., 0, (.0, 0.),.., '*xYz*',..}

Type univ@ is considered as the defining type for the other types of pscl, namely "int," "bool," "string," "cell@," "tuple@" from which all other types of CONLAN will be derived.

ANY@

Class **any@** is the universal class in CONLAN, and the only class known in pscl. Its domain is the set of designators for all types defined or yet to be defined in all members of the CONLAN family, together with operations '=', '≠', 'ϵ', and '<|'.

any@={univ@, int, bool, tuple@, cell@, string,...}

Function 'ϵ' may be used to determine if an object from univ@ is a member of a defined type (a member of any@).

Function '<|' may be used to determine if a member of any@ (a type) was derived from another member of any@.

INT

Type **int** consists of all integers, together with a substantial number of operators provided without formal definition, i.e., they are "known." An integer is denoted with a contiguous sequence of symbols, digits and capitals that may be partitioned into the sign part, magnitude part and base indicator. The sign part consists of symbol + (optional) or symbol −. The magnitude may be expressed in decimal, binary (B), octal (O), or hexadecimal (H). For instance,

$$-12=-1100B=-14O=-CH$$

BOOL

Type **bool** has two members, denoted by 1 and 0 representing "true" and "false" respectively, together with operations '=', '≠', '∧', '∨', '¬', '<', '≤', '>', '≥'. The relational operators are based upon 0 being less than 1.

STRING

Type **string** consists of all sequences of characters, together with operations '=', '≠', '<', '≤', '>', '≥', and order@. The objects of string are denoted by enclosing the sequence in single quotes ('). Sequences such as '1A', 'b+5', are included. The character ' must be doubled if it is to appear in a string denotation (e.g., 'What''s his name?').

The relational operators are based in the order of characters in the ISO-IRV 646 standard. When comparing strings of different length, the shorter string is padded or extended with trailing spaces (code 20H).

Function order@ takes a string as a parameter and returns an integer computed by treating the elements of the string (i.e., the characters) as 'digits' in a base 128 representation. The leading character is the most significant 'digit'. For instance, order@('*Xy*2') returns (58H*128*128+79H*128+32H), that is, 163CB2H.

TUPLE@

Type **tuple@** consists of all lists of members of univ@, together with operations '=', '≠', size@, select@, remove@, and extend@. Tuple@ includes the empty list. A tuple is denoted via a list of object denotations enclosed in '(.' and '.)', and separated by commas.

Two tuples are equal ('=') if they have the same size and identical members in identical order. Otherwise they are not equal ('≠').

Function size@(x) returns the number of members of a tuple. If the tuple is empty, size@ returns 0. Consecutive integers from 1 to size@(x) identify the positions of the members of tuple x. Only the positions of this range may be referenced. Attempts to reference positions outside this range result in an error report.

Function select@(x,i) returns the member of tuple x in position i (if integer i is in the range 1 through size@(x)).

Function remove@(x,i) returns the tuple y such that y holds all components of x in the same order except the ith one if i is in the range from 1 to size@(x) else an error condition exists. If size@(x) = 1 then the empty tuple is returned.

Function extend@(x,u) returns the tuple y of size size@(x)+1 with the leftmost size@(x) components being identical to those of x and u as the component in position size@(x)+1.

Tuples are never modified. Remove@ and extend@ simply select the member of the set of all tuples with the right size and contents. The original tuple is not altered.

CELL@

Type **cell@** consists of all 'containers' of members of univ@. Cells can contain at most one element of a type. The type of the contents must be specified as a parameter in a type_designator, when the cell is declared. For instance, cell@(t:int). Cells are initially empty.

Function cell_type@(x) returns a member of any@, namely the type of the (potential) contents of cell x.

Function empty@(x) returns 1 ('true') if cell x is empty otherwise it returns 0 ('false').

Function get@(x) returns the contents of cell x. If the cell x is empty an error condition exists.

Activity put@(x,u) replaces the contents of cell x and u. The type of u must be identical to the type of the contents of cell x. Attempts to put an element of the wrong type result in an error condition. If cell x is empty, put@ simply inserts u in the cell.

Cells are potentially modifiable objects (via function put@). These are the only objects with this attribute and constitute the bases for the development of carriers, variables, and other modifiable objects in the CONLAN family.

6. EXAMPLES OF TYPE AND CLASS DERIVATION

From the types of pscl new types have been derived using the techniques explained under type definition. For example,

6.1 Typed tuples

```
TYPE tytuple@(t: any@) BODY
  ALL x: tuple@ WITH FORALL i: bint(1,size@(x))
  IS@ x[i] ∈ t ENDFORALL ENDALL
  CARRY =, ≠, size@, remove@ ENDCARRY

  FUNCTION select(x: tytuple@(t), i: pint): t
    RETURN THE@ z: t WITH@ z =
    select@(old@(x),i)
  ENDselect

  FUNCTION tail(x: tytuple@(t)): tytuple@(t)
    ASSERT size@(x) > 1 ENDASSERT
    RETURN remove@(x,1)
  ENDtail

  FUNCTION extend(x: tytuple@(t), a: t) : tytuple@(t)
    RETURN new@(extend@(old@(x),a))
  ENDextend

                    "/Swap two elements of a typed tuple/"
  FUNCTION exchange(x: tytuple@(t), i, j: pint):
  tytuple@(t)
    ASSERT i ≤ size@(x), j ≤ size@(x) ENDASSERT
    RETURN THE@ y: tytuple@(t) WITH@
      size@(y) = size@(x) ∧
      FORALL@ k: bint(1,size@(x))) IS@
          IF k = i THEN y[k] = x[j]
```

```
        ELIF k = j THEN y[k] = x[i] ELSE y[k] = x[k]
        ENDIF ENDFORALL ENDTHE
  ENDexchange
ENDtytuple
```

Type tytuple@ is a generic type. Its formal parameter t may be bound to the designator of any defined type. In its definition two auxiliary types are used which are not explicitly derived here: (1) Type pint, whose domain is all positive integers together with all integer operations. and (2) Type bint (i,j:int), whose domain consists of all sets of consecutive integers $\{i,\ldots,j\}$.

The domain of type tytuple@ is the set of all tuples whose components are elements of the same type t. Functions $=$, $\neq$, size@, and remove@ are carried from tuple@ to tytuple@. In addition, three new functions are defined for elements of tytuple@: select, extend, exchange.

Function select(x:tytuple@(t),i:pint):t on tytuple@ is defined to be identical to function select@(x:tuple@,i:int):univ@ on tuple@ restricted to the types tytuple@, pint and t. A special type conversion operator old@(x) is used to convert the element x of tytuple@ to the old type tuple@ before the function select@ of tuple@ is applied.

Function extend on tytuple@ is equivalent to extend@ on tuple@ but restricted in the typing of its parameters and the result.

Function exchange forms a tuple y by interchanging elements i and j of tuple x.

6.2 Scalar value types

```
CLASS scalar_value_type BODY
  ALL x:any@ WITH x <| int ∨ x <| bool ∨ x <| string
  ENDALL
  CARRYALL
ENDscalar_value_type
```

Class scalar_value_type is defined as the set of those types (i.e., members of any@) which are equal to or derived from one of the primitive types int, bool, or string. The CARRYALL statement makes all relations defined on any@ also available to the types belonging to class scalar_value_type.

As an example of the use of classes:

```
TYPE scalar_cell(u:scalar_value_type) BODY
  cell@(u)
  CARRYALL
ENDscalar_cell
```

Generic type scalar_cell has a parameter u which may be bound only to a scalar_value_type, for example:

scalar_cell(bool)

specifies cells which may only contain Booleans. Thus writing

scalar_cell(tytuple(bool))

will result in a type checking error since because tytuple(bool) is not an element of the class scalar_value_type.

7. CONCLUSIONS

In this paper the motivation, objectives, basic concepts of the CONLAN family and its primitive set members are described.

To promote an orderly development of hardware description languages and to enhance their acceptance in an industrial environment, a powerful construction mechanism for such languages, based on a common core syntax is presented. This construction mechanism ensures that the semantics of the languages derived are well defined. Further, semantically related languages can be constructed which permit the description of digital systems at different levels of abstraction. The common core syntax facilitates learning a new language written in the CONLAN framework. In addition, capabilities for syntax modification permit the suppression of unneeded constructs and the introduction of shorthand for frequently used objects to obtain simple yet useful languages.

We feel that the primitive set language, together with the construction mechanism are not only valid for the development of hardware description languages but also represent a contribution to the available techniques for formal semantic specification of operative languages, including programming languages.

The three papers presented in this series illustrate the basic principles of the construction mechanism, the process of language derivation, and examples of the application of a derived language to hardware description.

Additional reports on the progress of the CONLAN working group are in preparation. Future efforts will be directed toward the preparation of a complete report covering our method for syntax modification, the formal development of the CONLAN array and record constructors, and Base CONLAN (the constructional base for user languages), as well as examples of user languages. In addition, the working group expects to develop more comprehensive user languages covering the design of systems at the gate level, register transfer level, instruction set level (macro and micro programming), and system level.

8. ACKNOWLEDGMENTS

The authors are indebted to Bell Northern Research (Ottawa), Sperry Univac (Philadelphia), Office of Naval Research, Ballistic Missile Defense Advanced Technical Center (Huntsville), IRIA (Paris), Busdesministerium fur Forschung und Technologie (Bonn), Siemens (Munich), and Fujitsu (Tokyo) for their interest and support, Professor Yaohan Chu for his early contributions, and in particular to Professor Jack Lipovski for his help and unwavering confidence in the group.

9. REFERENCES

1. Piloty, R., Barbacci, M., Borrione, D., Dietmeyer, D., Hill, F. and Skelly, P.,"CONLAN—A Formal Construction Method for Hardware Description Languages: Language Derivation," *Proceedings National Computer Conference*, Volume 49, Anaheim, California, 1980.
2. Piloty, R., Barbacci, M., Borrione, D., Dietmeyer, D., Hill, F. and Skelly, P., "CONLAN—A Formal Construction Method for Hardware Description Languages: Language Application," *Proceedings National Computer Conference*, Volume 49, Anaheim, California, 1980.
3. Barbacci, M. R., "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems," IEEE Computer Society, *Transactions on Computers*, Volume C-24, Number 2, February 1975.
4. Special issue on Hardware Description Languages, IEEE Computer Society, *Computer*, Vol. 7, No. 12, Dec. 1974.
5. *Proceedings of the 2nd International Symposium on Computer Hardware Description Languages*, Darmstadt, ACM German Chapter Lectures W—1974.
6. *Proceedings of the 3rd International Symposium on Computer Hardware Description Languages and their Applications*, New York, Sept. 3-5, 1975. IEEE Cat. No. 75 CH1010-8C.
7. *Proceedings of the 4th International Symposium on Computer Hardware Description Languages*, Palo Alto, Oct. 8-9, 1979 IEEE Cat. No. 79 CH1436-5C.
8. Special issue on Hardware Description Languages, IEEE Computer Society, *Computer*, Vol. 10, No. 6, June 1977.
9. Collection of Proceedings of the IEEE, ACM Design Automation Conference.
10. Collection of Proceedings of the Fault Tolerant Computing Symposia.
11. Piloty, R., "Guidelines for a Computer Hardware Description Consensus Language" (2nd draft), Memorandum to the Conference on Digital Hardware Languages, June 6, 1976.
12. Liskov, B. and Zilles, S., "Programming with Abstract Data Types," SIGPLAN Notices 9, pp. 50-59, April 1974.
13. Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C., "Abstraction Mechanism in CLU," Computation Structures Group, Memo 144-1, MIT January 1977.
14. Wulf, W. A., "Alphard: Toward a Language to Support Structured Programming," Technical Report, Department of Computer Science, Carnegie-Mellon University, April 1974.
15. Wulf, W. A., London, R. L. and Shaw, M., "Abstraction and Verification in ALPHARD," Technical Report, Department of Computer Science, Carnegie-Mellon University, March 1976.
16. Lucas, P. and Walk, K., "On the Formal Description of PL/I," *Annual Review of Automatic Programming*, Vol. 6, part 3, 1969.
17. Jacquet, P., "Les Types Generic: Propositions pour un Mecanisme d'Abstraction dans les Langages de Programmation."